СОДЕРЖАНИЕ

1 ТРЕБОВАНИЯ К РАБОТЕ

Практическая работа 1 (Многоагентное моделирование):

- Количество состояний агента – не менее 3;
- Количество параметров агента - не менее 5;
- Размер популяции агентов – не менее 500;
- Возможность динамического изменения параметров – не менее 1;
- Наличие условий в карте агента: не менее 1
- Наличие графика/ов, для отслеживания динамики изменения состояний популяции;

Практическая работа 2 (Дискретно-событийное моделирование):

- Количество состояний агента – не имеет значения;
- Количество параметров агента – не имеет значения;
- Размер популяции агентов – не менее 10;
- Возможность динамического изменения параметров – не менее 1;
- Наличие условий в карте агента: не менее 1
- Наличие картинки и 3D окна, для отслеживания динамики изменения состояний популяции;

Практическая работа 3 (Системная динамика):

- Количество накопителей – не менее 4;
- Количество потоков - не менее 3;
- Количество параметров агента – не менее 1;
- Возможность динамического изменения параметров – не менее 1;
- Наличие обратных связей – потоков (циклов);
- Наличие графика/ов, для отслеживания динамики изменения состояний популяции;

Практическая работа 4 (Работа по выбору):

- Выбрать одно задание из (практическая 7, 8, 9) – сделать программную реализацию задания из выбранной работы;

- Язык программирования – на свой выбор;
- Сделать визуализацию воспроизводимых процессов;
- (Опционально) Предложить свою тематику;

2 ВЫПОЛНЕНИЕ РАБОТЫ

2.1 Многоагентное моделирование

2.1.1 Введение

Многоагентное моделирование представляет собой метод имитационного моделирования, в котором система описывается как совокупность взаимодействующих агентов с индивидуальными правилами поведения. Такой подход позволяет изучать сложные явления, возникающие в результате локальных взаимодействий между агентами. Многоагентные модели широко применяются в экономике, экологии, социологии и других областях. В данной работе рассматривается построение и анализ многоагентной модели с целью исследования динамики системы при различных значениях параметров.

2.1.2 Общее описание

В данной работе представлено моделирование распространения двух вирусов “Carnage” и “Venom”, агент может приобрести один из этих вирусов. После заражения, оба вируса могут мутировать в новый вирус “SuperVenom” если агент будет взаимодействовать с обладателями противоположного вируса, либо агент может умереть, если будет взаимодействовать с зараженными агентами, либо слишком долго будет заражён одним вирусом.

2.1.3 Техническое описание

В данной работе из состояния “Zdorovi” можно перейти в состояния “Venom” и “Carnage” с заданной интенсивностью, в среде Anylogic состояния подразумевают под собой область, в которую переходит объект при определённых факторах. Интенсивности вычисляются как *zabolV * main.zarazV* и *zabolC * main.zarazC* соответственно, где *zabolV и zabolC* константы равные

0,01, а *main.zarazV* и *main.zarazC* глобальные переменные, которые регулируются с помощью ползунков, во время моделирования. Либо можно перейти в состояние “Smert” если агент получает сообщение “Kill”. Также из состояний “Venom” и “Carnage” осуществляется переход в “Zdorovi” с заданной интенсивностью равной 0,35.

Состояния “Venom” и “Carnage” с заданной интенсивностью отправляют сообщения “Venom” и “Carnage” соответственно, если агент с состоянием “Venom” получает сообщение “Carnage”, и наоборот, то агент переходит в состояние “SuperVenom”. Также агенты отправляют сообщения “Kill”. Интенсивность контролируется с помощью глобальных переменных, которые также можно контролировать с помощью ползунков, во время моделирования. Также из этих состояний с заданной интенсивностью, или при выполнении условия(а именно перехода глобальной переменной в 1), можно перейти в состояние “Smert”, интенсивность контролируется с помощью константной переменной *smert* равной 0,005.

Из состояния “Smert” можно перейти только в состояние “Zdorovi” только с помощью перехода глобальной переменной *voskres* в 1.

Из состояния “SuperVenom” можно перейти только в состояние “ Smert ” только с помощью перехода глобальной переменной *killS* в 1. Также есть ветвление, которое позволяет перейти в состояния “Venom” и “Carnage”.

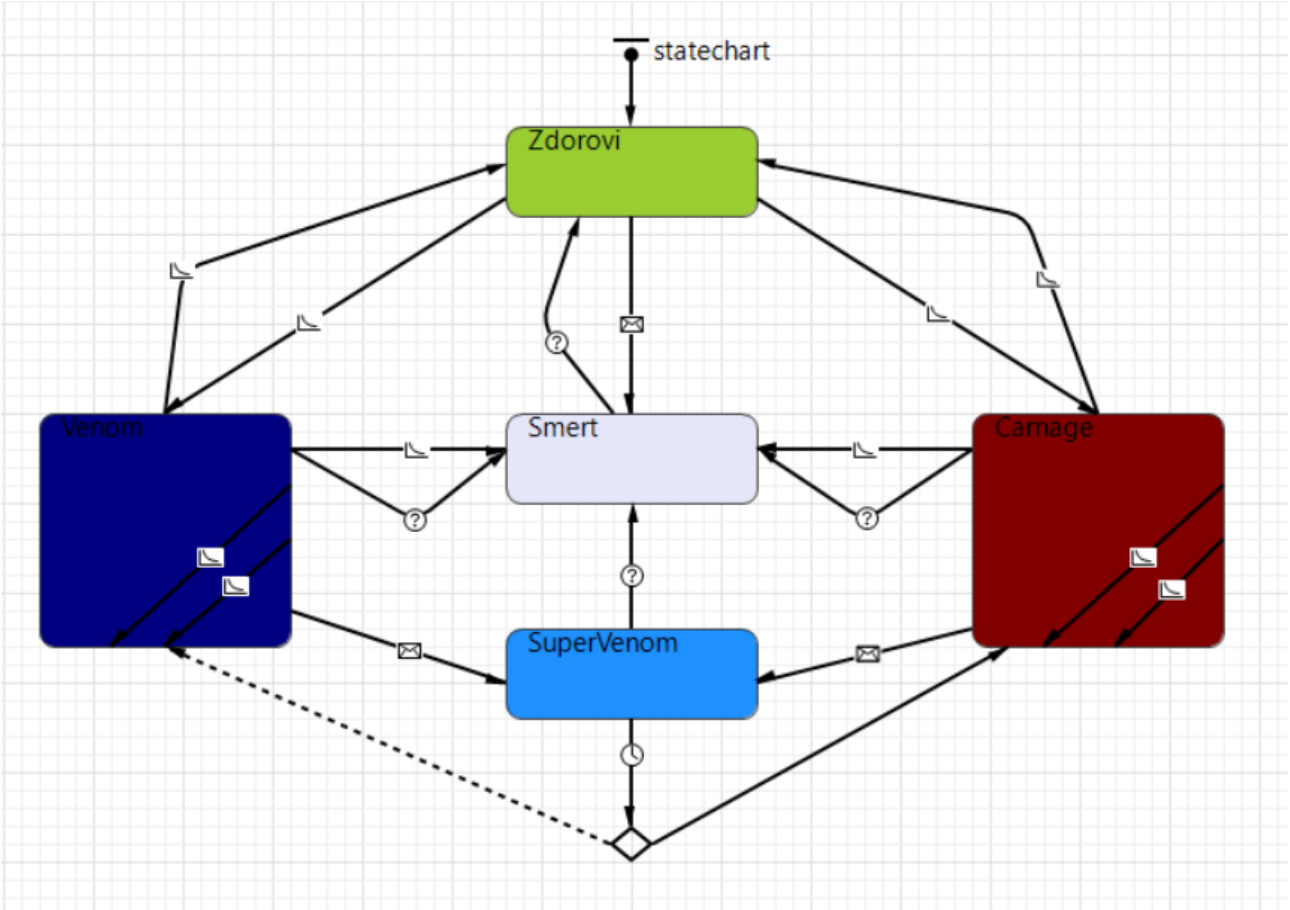

Рисунок 2.1.1 – Схема переходов состояний агента

2.1.4 Пример работы

Приведём пример выполнения данной работы.

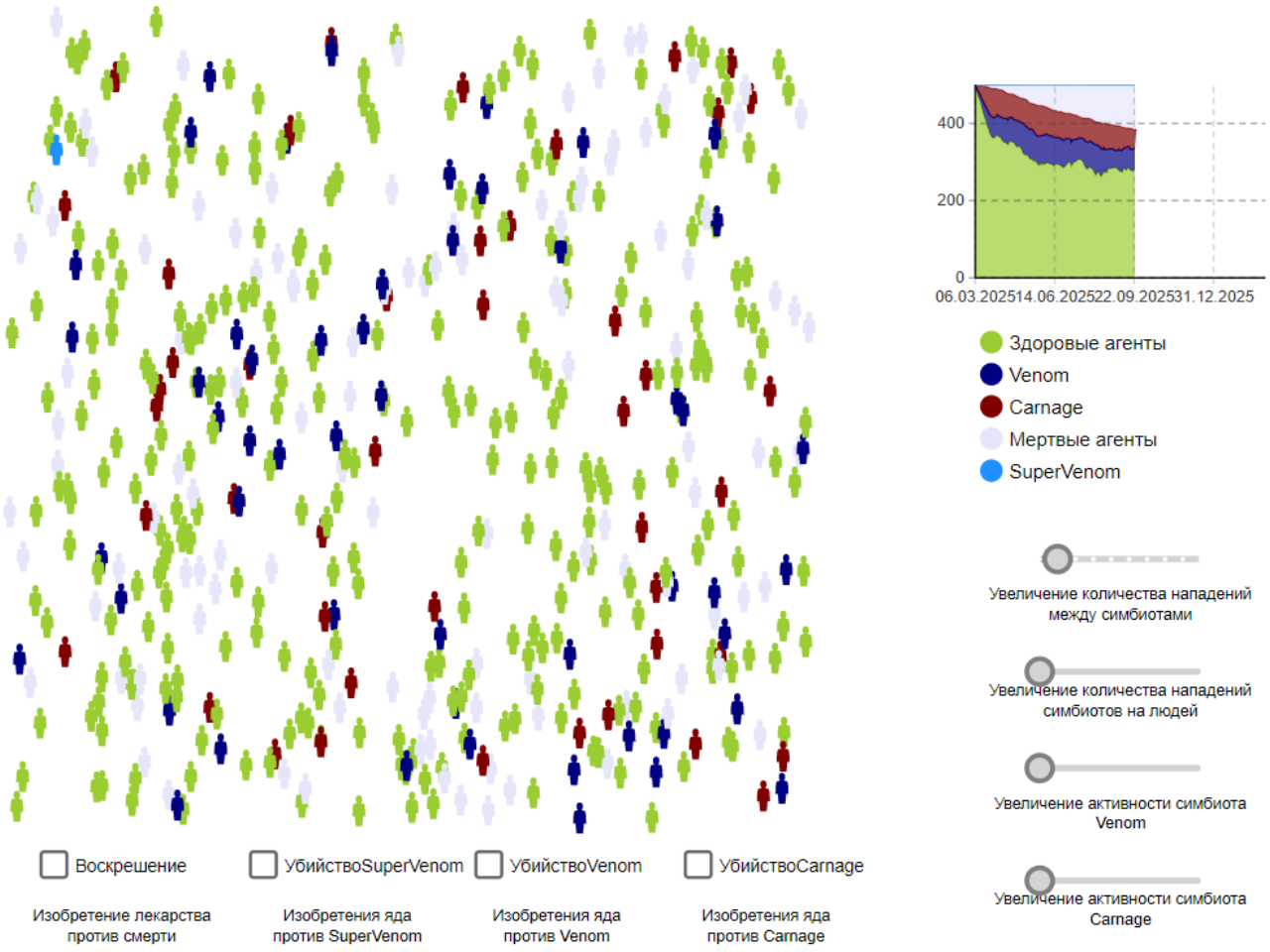

Рисунок 2.1.2 – Пример выполнения работы

2.2 Дискретно-событийное моделирование

2.2.1 Введение

Дискретно-событийное моделирование (ДСМ) — это метод имитационного моделирования, в котором поведение системы представляется как последовательность отдельных событий, происходящих во времени. Каждое событие мгновенно изменяет состояние системы, что позволяет точно отслеживать динамику процессов. ДСМ широко используется для анализа систем обслуживания, логистики, производственных процессов и компьютерных сетей. В данной работе рассматривается построение модели с использованием ДСМ для изучения поведения системы при различных условиях.

2.2.2 Общее описание

В данной работе представлено моделирование разработки, сортировки и перенос готовых двигателей. Первоначально на конвейере появляется деталь, после манипулятор передаёт её на рабочую поверхность, где роботом собирается двигатель, после чего двигатель возвращается на конвейер. Далее деталь перемещается на стеллаж, с которого вором перемещается в машину.

2.2.3 Техническое описание

Основным агентом в данной системе является агент MachinePart, состояния взяты из библиотек производственных сетей и моделирования процессов среды Anylogic. Из source агент попадает в блок conveyorIn, после выполнения этого события (передвижение по конвейерной ленте), начинается комплексное событие по сбору детали. Сначала происходит событие передачи детали (toMachine), после происходит обработка детали (weldRight, weldLeft) и перенос детали Cover (processByRobot), с последующим объединением в блоке combine и возвращение на конвейер (toConveyor). В блоке conveyorExit агент уходит с конвейера и переносится на стеллаж, с которого его переносят с помощью ресурсов vor.

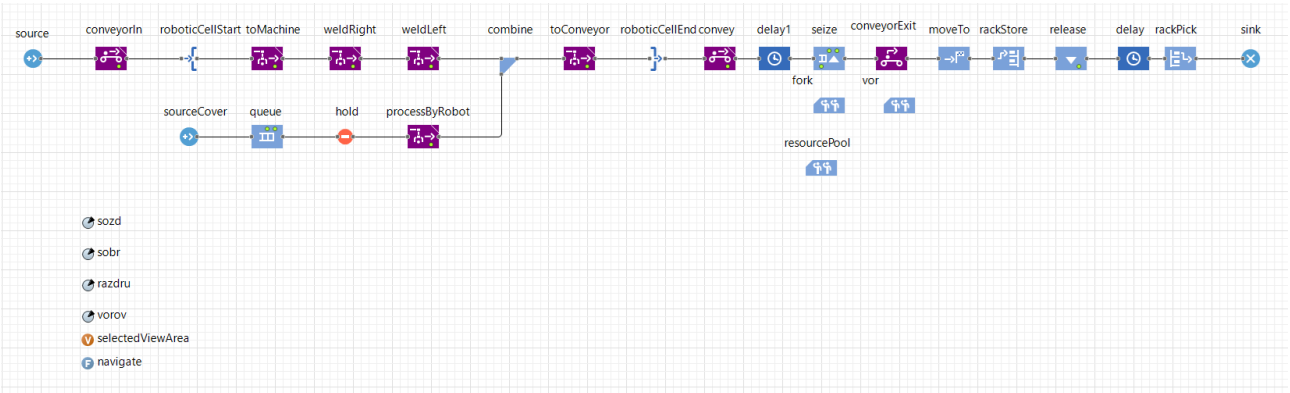

Рисунок 2.2.1 – Последовательность событий

2.2.4 Пример работы

Приведём пример выполнения данной работы.

Рисунок 2.2.2 – Пример работы (1/3 часть)

Рисунок 2.2.3 – Пример работы (2/3 часть)

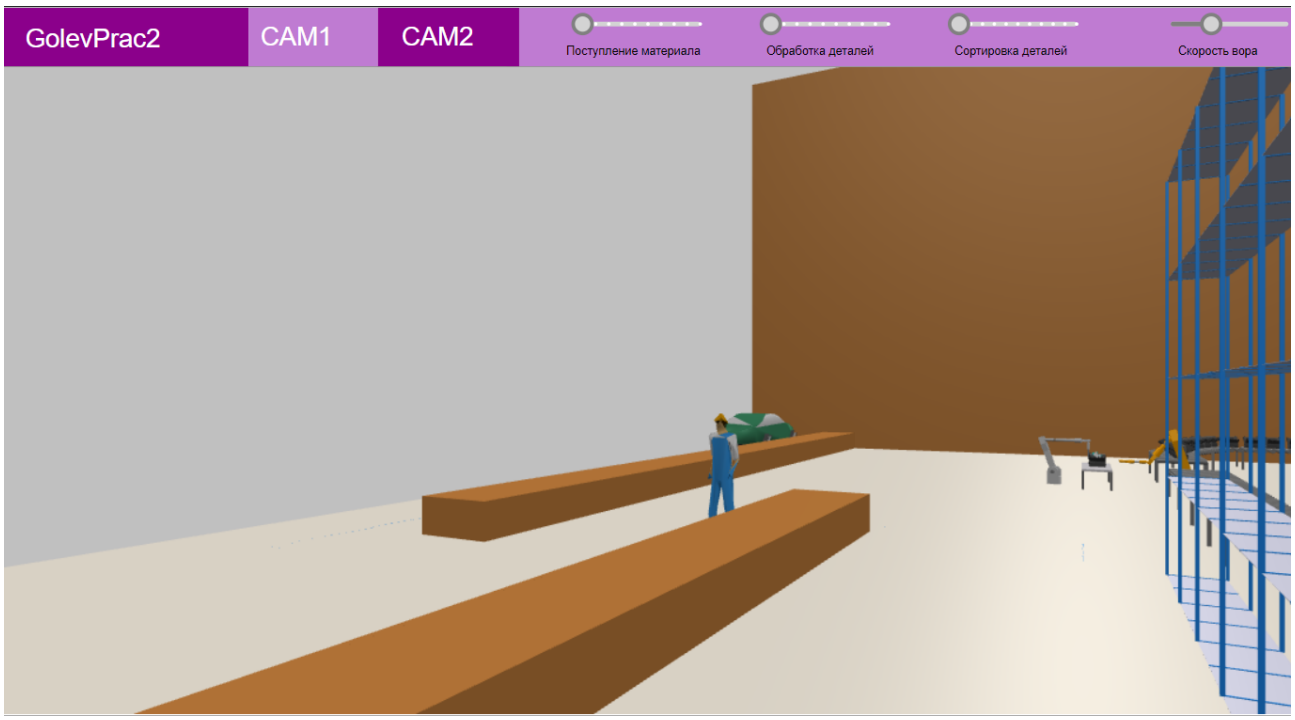

Рисунок 2.2.3 – Пример работы (3/3 часть)

2.3 Системная динамика

2.3.1 Введение

Системная динамика — это метод моделирования и анализа сложных систем, основанный на изучении потоков, накоплений и обратных связей между элементами системы. Она позволяет выявлять поведение системы во времени и оценивать влияние различных факторов на её развитие. Метод широко применяется в управлении, экономике, экологии и социальных науках. В данной работе рассматривается применение системной динамики для построения модели и анализа её поведения при изменении параметров.

2.3.2 Общее описание

В данной представлено моделирование превращение нефти в золото. Превращение происходит двумя путями – перегонка и магия. Перегонка – нефть превращается в бензин и продается. Магия – нефть путём крекинга превращается в мазут, после с помощью керосин в мазут и с помощью алхимии в золото.

2.3.3 Техническое описание

Имеется 5 накопителей, все элементы представленные в работе взяты из билиотеки системной динамики среды AnyLogic. Начальный накопитель SirNeft имеет начальное значение равное значению переменной TotalVolume. Из SirNeft в Benzin передаётся с формулой *MoshnPechki*SirNeft/TotalVolume,* и из Benzin передаётся в Zoloto с помощью формулы *Benzin*EffecivRinka*. Из SirNeft в Mazyt передаётся с формулой *Davlenie*SirNeft/TotalVolume,* из Mazyt передаётся в Kerosin с помощью формулы *Mazyt/(1001-Manna),* и из Kerosin передаётся в Zoloto с помощью формулы *Kerosin/(1010-PolezMat)*. Также из Zoloto передаётся в SirNeft с помощью формулы *Zoloto/((1010-Manna)*2)*.

2.3.4 Пример работы

Приведём пример выполнения данной работы.

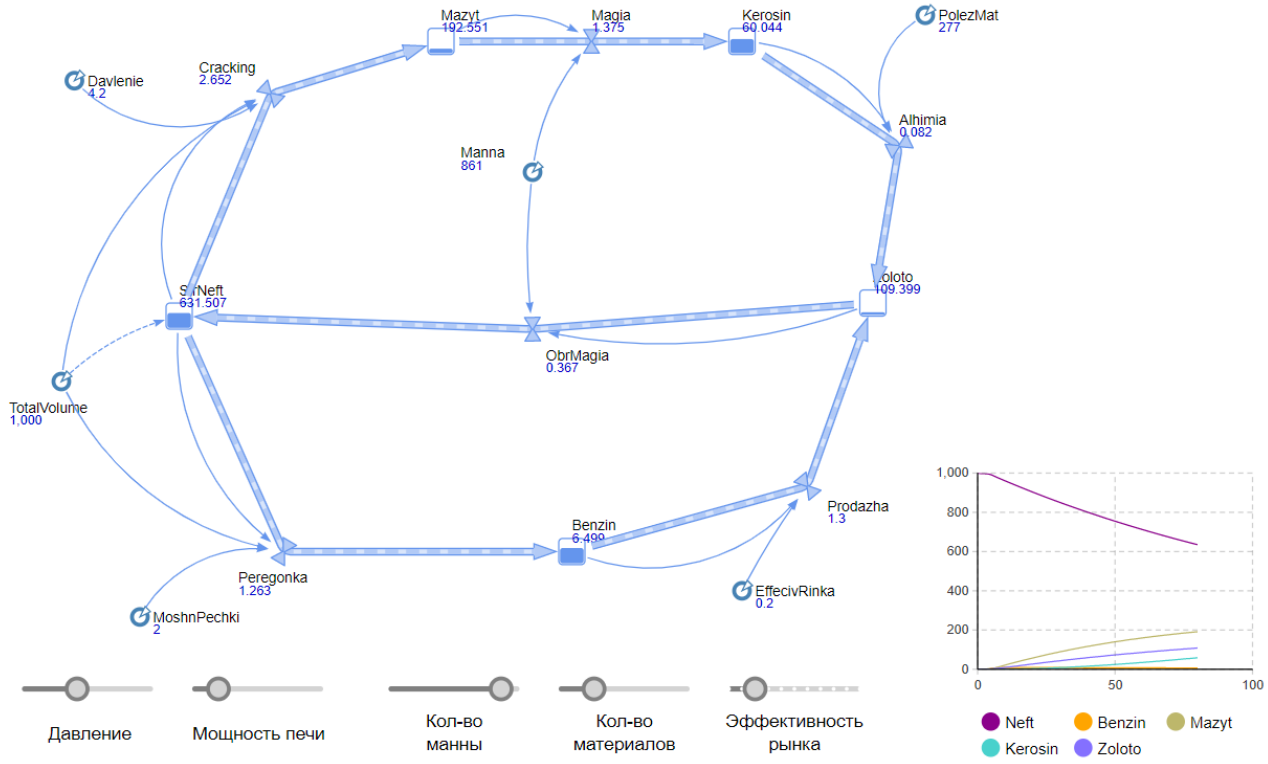

Рисунок 2.3.1 – Пример работы

2.4 Работа по выбору

2.4.1 Введение

В данной работе выбрано задние из практической работы 1, но реализовано программно на языке Python.

2.4.2 Общее описание

Общее описание соответсвует практической работе 1.

2.4.3 Техническое описание

В данной работе агенты представлены в виде агентов класса Agent. Имеют метод, который принимает сообщения от других агентов, и имеет переменную состояния. Все действия, совершаемые агентами, соответствуют действиям агентов из практики 1. Кодовая реализация представляет под собой два основных класса, Agent – класс, объекты которого являются агентами в работе, Simulation – класс, объект которого холст. Холст имеет в себе изображение агентов в виде матрицы, ползунки и кнопки с помощью которых можно влиять на моделирования и диаграмма, показывающая количество агентов в различных состояниях. Код программы представлен в приложении А.

2.4.4 Пример работы

Приведём пример выполнения данной работы.

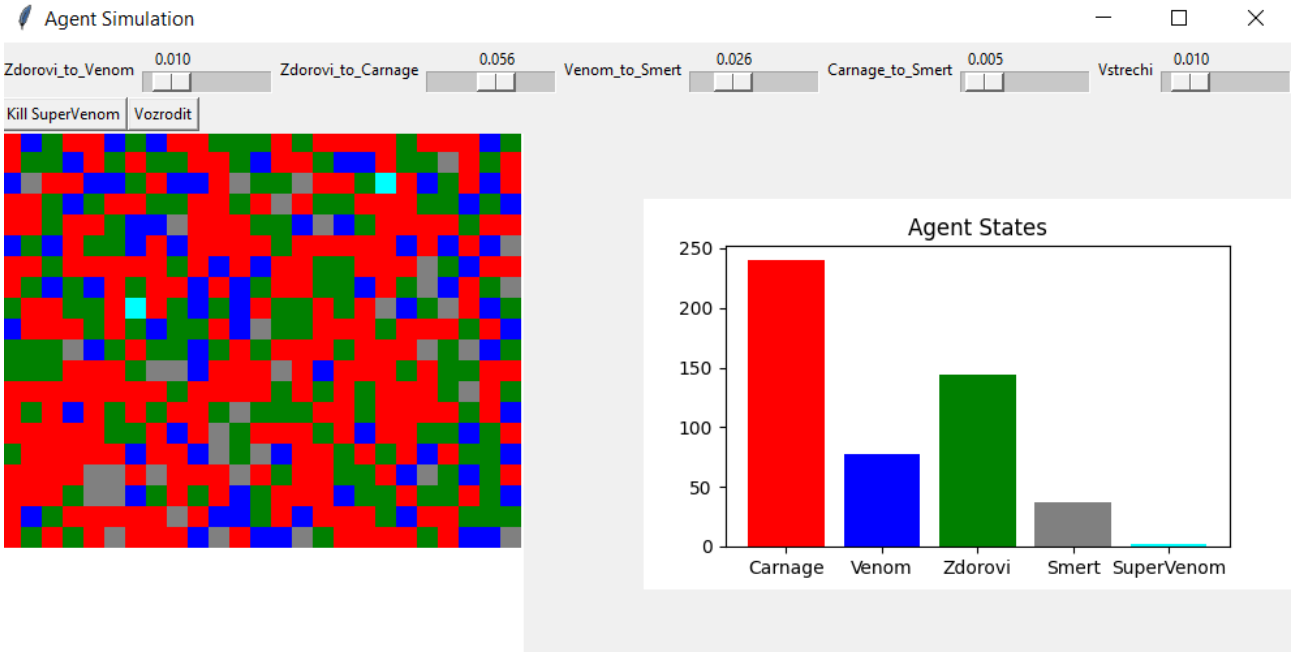

Рисунок 2.4.1 – Пример работы

3 ВЫВОД

В результате рассмотрения трёх методов имитационного моделирования — многоагентного моделирования, дискретно-событийного моделирования и системной динамики — можно отметить, что каждый из них обладает своими сильными сторонами и наилучшей областью применения. Многоагентное моделирование позволяет детально описывать поведение индивидуальных элементов и их взаимодействия, дискретно-событийный подход эффективен для анализа процессов с чёткой последовательностью событий, а системная динамика — для исследования глобальных структур и обратных связей в системе. Выбор метода зависит от задач моделирования, уровня детализации и характера исследуемой системы. Комплексное понимание этих подходов расширяет возможности анализа и разработки более точных и адекватных моделей.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Многоагентное моделирование. : учебно-методическое пособие / В. И. Тихвинский, В. В. Холмогоров, В. А. Морозов [Электронный ресурс].— М. : РТУ МИРЭА , 2022

ПРИЛОЖЕНИЯ

Приложение А – Код реализации практики 4 на языке Python

Приложение А

Код реализации практики 4 на языке Python

Листинг А.1. Реализация МАИ.

```
import tkinter as tk
import random
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from collections import Counter

STATES = ['Zdorovi', 'Venom', 'Carnage', 'SuperVenom', 'Smert']
COLOR_MAP = {
    'Zdorovi': 'green',
    'Venom': 'blue',
    'Carnage': 'red',
    'SuperVenom': 'cyan',
    'Smert': 'gray'
}
class Agent:
    def __init__(self):
        self.state = 'Zdorovi'
    def receive_message(self, message):
        if message == 'Kill' and self.state == 'Zdorovi':
            self.state = 'Smert'
        elif message == 'Venom' and self.state == 'Carnage':
            self.state = 'SuperVenom'
        elif message == 'Carnage' and self.state == 'Venom':
            self.state = 'SuperVenom'
class Simulation:
    def __init__(self, root):
        self.root = root
        self.agents = [Agent() for _ in range(500)]
        self.intensities = {
            'Zdorovi_to_Venom': 0.01,
            'Zdorovi_to_Carnage': 0.01,
            'Venom_to_Smert': 0.005,
            'Carnage_to_Smert': 0.005,
            'Vstrechi': 0.01
        }
        self.setup_gui()
        self.update_simulation()
```

Листинг А.2. Продолжение листинга А.1.

```
      def setup_gui(self):
              control_frame = tk.Frame(self.root)
              control_frame.pack(side=tk.TOP, fill=tk.X)

              # Slid
              for key in self.intensities:
                  lbl = tk.Label(control_frame, text=key)
                  lbl.pack(side=tk.LEFT)
                  slider = tk.Scale(control_frame, from_=0, to=0.1,
resolution=0.001,
                                    orient=tk.HORIZONTAL,
command=self.update_intensity(key))
                  slider.set(self.intensities[key])
                  slider.pack(side=tk.LEFT)
              # Butt
              btn_frame = tk.Frame(self.root)
              btn_frame.pack(side=tk.TOP, fill=tk.X)
              btn_sv_to_sm = tk.Button(btn_frame, text="Kill SuperVenom",
command=self.kill_supervenom)
              btn_sv_to_sm.pack(side=tk.LEFT)
              btn_sm_to_zd = tk.Button(btn_frame, text="Vozrodit",
command=self.revive_smert)
              btn_sm_to_zd.pack(side=tk.LEFT)
              # Canvas
              self.canvas = tk.Canvas(self.root, width=400, height=400,
bg='white')
              self.canvas.pack(side=tk.LEFT)
              # Graph
              self.fig, self.ax = plt.subplots(figsize=(5, 3))
              self.graph_canvas = FigureCanvasTkAgg(self.fig,
master=self.root)
              self.graph_canvas.get_tk_widget().pack(side=tk.RIGHT)
          def update_intensity(self, key):
              def update(val):
                  self.intensities[key] = float(val)
        return update
      def kill_supervenom(self):
              for agent in self.agents:
                  if agent.state == 'SuperVenom':
                agent.state = 'Smert'
```

Листинг А.3. Продолжение листинга А.2.

```
      def revive_smert(self):
              for agent in self.agents:
                  if agent.state == 'Smert':
                      agent.state = 'Zdorovi'
          def update_simulation(self):
              for agent in self.agents:
                  if agent.state == 'Zdorovi':
                      if random.random() <
self.intensities['Zdorovi_to_Venom']: agent.state = 'Venom'
                      elif random.random() <
self.intensities['Zdorovi_to_Carnage']: agent.state = 'Carnage'
                  elif agent.state == 'Venom':
                      if random.random() < self.intensities['Venom_to_Smert']:
                          agent.state = 'Smert'
                      if random.random() < self.intensities['Vstrechi']:
                          target = random.choice(self.agents)
                          target.receive_message(random.choice(['Kill',
'Venom']))
                  elif agent.state == 'Carnage':
                      if random.random() <
self.intensities['Carnage_to_Smert']: agent.state = 'Smert'
                      if random.random() < self.intensities['Vstrechi']:
                          target = random.choice(self.agents)
                          target.receive_message(random.choice(['Kill',
'Carnage']))
              self.update_canvas()
              self.update_graph()
              self.root.after(100, self.update_simulation)
          def update_canvas(self):
              self.canvas.delete('all')
              cols = 25
              size = 400 // cols
              for idx, agent in enumerate(self.agents):
                  row = idx // cols
                  col = idx % cols
                  x0, y0 = col*size, row*size
                  x1, y1 = x0+size, y0+size
                  color = COLOR_MAP.get(agent.state, 'white')
                  self.canvas.create_rectangle(x0, y0, x1, y1, fill=color,
outline='')
```

Листинг А.4. Продолжение листинга А.3.

```
      def update_graph(self):
              self.ax.clear()
              counts = Counter(agent.state for agent in self.agents)
              self.ax.bar(counts.keys(),  counts.values(),  color=[COLOR_MAP[s]
for s in counts.keys()])
              self.ax.set_title("Agent States")
              self.graph_canvas.draw()


      if __name__ == '__main__':
          root = tk.Tk()
          root.title("Agent Simulation")
          app = Simulation(root)
    root.mainloop()
```